TIE-20100 Tietorakenteet ja algoritmit, kevät 2018

Harjoitustyö 3:
Roads are coming

Viimeksi päivitetty 30.04.2018

Huom. koska tämä harjoitustyö pohjautuu aiempaan harjoitustyöhön, on allaolevassa merkitty harmaalla pohjalla kaikki muuttuneet / uudet asiat.

Harjoitustyön aihe

Harjoitustyön yhteyskunta on kehittynyt ja verotuksen lisäksi keksinyt tieverkoston. Kaupunkien välille voi rakentaa teitä ja olemassaolevia teitä voi tuhota. Ohjelmalla voi selvittää erilaisia reittejä teitä käyttäen ja optimoida tieverkostoa.

HUOM! *Tässä harjoitustyössä arvostellaan ainoastaan työn uudet operaatiot (esim. verotukseen liittyviä operaatioita ja nimien aakkostusta ei arvostella enää, koska ne arvosteltiin jo aiemmissa töissä). Luonnollisesti kuitenkin uusien operaatioiden vaatimat perustoiminnot (kaupunkien lisäys, hakeminen yms.) ovat edelleen mukana arvostelussa, erityisesti ne osat joihin on täytynyt tehdä muutoksia vanhojen operaatioiden tukemiseksi.*

Teiden lisäämisen ja poistamisen lisäksi ohjelmassa on seuraavanlaisia operaatioita:

- Kaikkien teiden tulostaminen, tietysti kaupungista lähtevien teiden tulostaminen
- Mielivaltaisen reitin etsiminen kahden annetun kaupungin välillä
- Sellaisen reitin etsiminen kahden kaupungin välillä, jossa reitin varrella on mahdollisimman vähän kaupunkeja (kuvitellaan, että kaupungit vaikkapa ottavat maksua niiden läpi kulkemisesta)
- Pituudeltaan mahdollisimman lyhyen reitin etsiminen kahden kaupungin välillä
- Rengasreittien etsiminen
- "Tarpeettomien" teiden tuhoaminen niin, että kaupunkien saavutettavuus ei muutu, mutta tieverkoston kokonaispituus minimoidaan (ylläpitokustannusten karsimiseksi)

Toisessa harjoitustyössä ensimmäisen harjoitustyön ohjelmaa laajennettiin niin, että se osaa käsitellä myös kaupunkien verotussuhteita. Historian edetessä kuvitteellisen maailmamme kaupungit ovat alkaneet tehdä yhteistyötä, ja vahvimmat kaupungit tarjoavat tukeaan heikommille, mutta vaativat vastapainoksi osan ko. kaupungin verokertymästä.

Kaupunkien yksilöiminen tehdään nyt yksikäsitteisen ID-tunnisteen avulla (osoittimien sijaan). Harjoitustyössä paino on valmiiden tietorakenteiden ja algoritmien käytössä (STL), mutta edelleen harjoitellaan myös algoritmien toteuttamista ja niiden tehokkuuden arvioimista.

Harjoitustyössä ohjelmaan syötetään tietoja kaupungeista (nimi ja koordinaatit ja verokertymä), ja ohjelmalta voi kysyä kaupunkeja halutussa järjestyksessä sekä minimi- ja maksimitietoja. Ei-

pakollisena osana kaupunkeja voi myös poistaa. Kysyä voi myös kaupunkien verotussuhteita ja verokertymää.

Koska kyseessä on Tietorakenteiden ja algoritmien harjoitustyö, ohjelman tehokkuus on tärkeä arvostelukriteeri. Tavoitteena on tehdä mahdollisimman tehokkas toteutus, kun oletetaan että kaikki ohjelman tuntemat komennot ovat suunnilleen yhtä yleisiä (ellei komentotaulukossa toisin mainita). Plussaa tietysti saa, mitä tehokkaammin operaatiot pystyy toteuttamaan.

Huomaa erityisesti seuraavat asiat:

- Tässä harjoitustyössä *testataan ja arvostellaan vain uuteen toiminnallisuuteen liittyvät osat*, eli uudet operaatiot ja vanhoista operaatioista ne osat, jotka niihin on lisättävä tai muutettava uuden toiminnallisuuden tukemiseksi.
- Tässä harjoitustyössä harjoitellaan edelleen valmiiden tietorakenteiden ja algoritmien (STL) tehokasta käyttöä, joten kannattaa suosia STL:ää omien algoritmien/tietorakenteiden sijaan silloin, kun se on tehokkuuden kannalta järkevää.
- Kolmannessa harjoitustyössä pakollisia uusia operaatioita ovat all_roads(), get_roads_from(), add_road(), remove_road(), clear_roads() ja any_route(). Operaatioiden least_towns_route(), shortest_route(), road_cycle_route() ja trim_road_network() toteuttaminen ei ole pakollista läpipääsyn kannalta. Ne ovat kuitenkin osa arvostelua, joten toteuttamatta jättäminen vaikuttaa työn arvosanaan! *Maksimiarvosana ilman yhtään ei-pakollista osaa on 2.*
- Vinkki ei-pakollisten operaatioiden toteuttamiseen. Kurssin pitäjän arvaus on, että least_roads_path() on helpoin toteuttaa, sitten tulee find_road_cycle() ja shortest_path() ja vaikein lienee trim_road_network(). Tämän viimeisen operaation osalta joudut itse hieman miettimään ja etsimään sopivaa algoritmia.
- Kolmannen harjoitustyön operaatioissa asymptoottiseen tehokkuuteen ei välttämättä pysty hirveästi vaikuttamaan, koska käytetyt algoritmit määräävät sen. Sen vuoksi kolmannessa harjoitustyössä algoritmien toteutukseen ja toiminnallisuuteen kiinnitetään enemmän huomiota kuin tehokkuuteen. Kurssin puolesta tulee kuitenkin tehokkuustestejä, erillinen testi jokaiselle etsintäoperaatiolle.
- Valmiina annetun pääohjelman voi ajaa joko graafisen käyttöliittymän kanssa QtCreatorilla/qmakella käännettynä, tai tekstipohjaisena pelkällä g++:lla käännettynä. Itse ohjelman toiminnallisuus ja opiskelijan toteuttama osa on täsmälleen sama molemmissa tapauksissa.
- **Vihje** tehokkuudesta: Jos minkään operaation keskimääräinen tehokkuus on huonompi kuin $\Theta(n \log n)$,ratkaisun tehokkuus *ei* ole hyvä. Suurin osa operaatioista on mahdollista toteuttaa paljon nopeamminkin. Kolmannen harjoitustyön jotkin operaatiot saattavat pakosti olla hitaampiakin.
- **Osana ohjelman palautusta tiedostoon datastructures.hh on jokaisen operaation oheen laitettu kommentti, johon lisätään oma arvio kunkin toteutetun operaation asymptoottisesta tehokkuudesta lyhyiden perusteluiden kera. Näitä kommentteja**

tarvitsee kolmannen harjoitustyön osalta päivittää vain uusien ja muuttuneiden asioiden osalta.

- **Osana ohjelman palautusta palautetaan git:ssä myös dokumentti nimeltä "readme.pdf" (samassa hakemistossa/kansiossa kuin lähdekoodi), jossa perustellaan toteutuksessa käytetyt tietorakenteet tehokkuuden kannalta. Tätä dokumenttia tarvitsee kolmannen harjoitustyön osalta päivittää vain uusien ja muuttuneiden asioiden osalta.**
- Operaatioiden `remove()`, `towns_distance_increasing_from()`, `longest_vassal_path()` ja `total_net_tax()` toteuttaminen ei ole pakollista läpipääsyn kannalta. ~~Ne ovat kuitenkin osa arvostelua, joten toteuttamatta jättäminen vaikuttaa harjoitustyön arvosanaan!~~ **~~Vain pakolliset osat toteuttamalla harjoitustyön maksimiarvosana on 3.~~**
- Tehokkuudessa olennaisinta on, miten ohjelman tehokkuus muuttuu datan kasvaessa, eivät pelkät sekuntimäärät.
- Operaation tehokkuuteen lasketaan kaikki siihen liittyvä työ, myös mahdollisesti alkioiden lisäyksen yhteydessä tehty operaation hyväksi liittyvä työ.
- Plussaa tietysti saa, jos operaatioita saa toteutettua vaadittua mahdollisimman tehokkasti.
- Samoin plussaa saa, mitä nopeammaksi operaatiot saa sekunteinakin (jos siis kertaluokka on vähintään vaadittu). **Mutta** plussaa saa vain tehokkuudesta, joka syntyy omista algoritmivalinnoista ja suunnittelusta. (Esim. kääntäjän optimointivipujen väänteły, rinnakkaisuuden käyttö, häkkerioptimoinnilla kellojaksojen viilaaminen eivät tuo pisteitä.)
- Riittävän huonolla toteutuksella työ voidaan hylätä.
- Esimerkkejä kysymyksistä, joilla tehokkuutta voi usein parantaa: Tehdäänkö jokin asia turhaan useaan kertaan? Voiko jonkin asian joskus jättää kokonaan tekemättä? Tehdäänkö joissain työtä enemmän kuin on välttämättä tarpeen? Voiko jonkin asian tehdä "lähes ilmaiseksi" samalla, kun tehdään jotain muuta?

Kaupunkien etäisyyksistä

Joissain operaatioissa vertaillaan kaupunkien etäisyyksiä (joko origosta (0,0) tai annetusta koordinaatista). Jotta automaattitesteissä ei jouduttaisi ongelmiin liukulukujen kanssa, sovitaan että tämän harjoitustyön maailmassa on mahdollista liikkua vain yksi yksikkö kerrallaan itään, länteen, pohjoiseen tai etelään (ei siis vinoon). Tällöin kahden pisteen välisen etäisyyden voi laskea yksinkertaisesti kaavalla $|x1-x2|+|y1-y2|$, eli koordinaattien erotusten itseisarvon summana (kiinnostuneille, tällaista geometriaa kutsutaan nimellä *Manhattan/taxicab geometry*).

Reittien tulostamisesta

Harjoitustyössä operaatiot palauttavat usein reitin kahden kaupungin välillä. Kooditasolla tämä tapahtuu palauttamalla lista kaupunki-id:itä niin, että lähtökaupunki on ensimmäisenä, sitten reitin seuraava kaupunki jne., ja päämäärä viimeisenä. Annettu pääohjelma tulostaa palautetun reitin niin,

että se listaa kaikkien reitin varrella olevien kaupunkien tiedot, ja lisäksi näyttää joka kaupungin kohdalla reitin senhetkisen pituuden (tulostuksessa "->" merkin jälkeinen luku).

Verokertymän laskemisesta

Harjoitustyössä kaupungit maksavat toisille kaupungeille veroa. Kaupunkia, joka maksaa veroa toiselle kaupungille, kutsutaan veroja saavan kaupungin *vasallikaupungiksi (vassal town).*Vastaavasti veroja vasallikaupungilta saavaa kaupunkia kutsutaan *isäntäkaupungiksi (master town).*

Jokaiselle kaupungille määritellään kaupunkia lisättäessä sen vuotuinen kansalaisilta saatava vero (kokonaislukuna). Tämän lisäksi kaupunki saa 10 % jokaisen vasallikaupunkinsa kokonaisverokertymästä (mukaanlukien vasallikaupungin mahdollisesti omilta vasalleiltaan saamat verot). Tästä kertymästä (kansalaisilta saatava määrä + vasalleilta saatava kymmenys) kaupunki maksaa sitten vielä veroa omalle isäntäkaupungilleen (jos sellainen on). Laskuissa 10 % laskeminen pyöristetään aina alaspäin (kuten C++:n kokonaislukujen jakolasku tekee), ja tämä alas pyöristetty summa vähennetään sitten verokertymästä. Tuloksena saatavaa lukua kutsutaan tässä *nettoverokertymäksi (total net tax).*

Esimerkki: kaupungilla *x* (vero 20) on kaksi vasallia *v1* (vero 10) ja *v2* (vero 5), joilla ei ole omia vasalleja. Lisäksi kaupungilla *x* on isäntäkaupunki *i.* Tällöin kaupungin *v1* nettoverokertymä on 9 (koska 10 % maksetaan verona *x*:lle), kaupungin *v2* nettoverokertymä on 5 (koska 10 % 5:stä pyöristyy nollaksi). Kaupungin *x* nettoverokertymä on 19 (20 + v*1*:ltä saatava 1 (ja v*2*:lta saatava 0) = 21, josta 10 % eli pyöristettynä 2 maksetaan *i*:lle, tekee yhteensä 19).

Järjestämisestä

Kaupunkeja järjestettäessä on mahdollista, että etäisyyden mukaan järjestettäessä usealla on sama etäisyys (tai nimijärjestyksessä nimi). Tällaisten tapausten keskinäinen järjestys on mielivaltainen.

Ohjelman hyväksymissä nimissä voi olla vain kirjaimia A-Z ja a-z. Järjestämisen voi tehdä joko std::string-luokan vertailuoperaattorin "<" mukaan (jossa isot kirjaimet tulevat ennen pieniä) tai "oikealla tavalla", jossa vastaavat isot ja pienet kirjaimet ovat samanarvoisia.

Kaupunkien ID:iden järjestämisessä tulee käyttää C++:n string-luokan "<"-vertailua.

Harjoitustyön toteuttamisesta ja C++:n käytöstä

Harjoitustyön kielenä on C++14. Tämän harjoitustyön tarkoituksena on opetella valmiiden tietorakenteiden ja algoritmien käyttöä, joten C++:n STL:n käyttö on erittäin suotavaa ja osa arvostelua. Mitään erityisiä rajoituksia C++:n standardikirjaston käytössä ei ole. Luonnollisesti kielen ulkopuolisten kirjastojen käyttö ei ole sallittua (esim. Windowsin omat kirjastot tms.).

HUOMAA, että koska tämän harjoitustyön tarkoituksena on harjoitella STL:n käyttöä, on *erittäin todennäköistä*, että ensimmäisen harjoitustyön tietorakenneratkaisusi EI ole paras mahdollinen tässä harjoitustyössä. Samoin kannattaa harkita, missä kohdin ensimmäisessä harjoitustyössä itse toteutettuja algoritmeja voi korvata STL:n valmiilla algoritmeilla!

Ohjelman toiminta ja rakenne

Osa ohjelmasta tulee valmiina kurssin puolesta, osa toteutetaan itse.

Valmiit osat, jotka tarjotaan kurssin puolesta

Tiedostot *mainprogram.hh, mainprogram.cc, mainwindow.hh, mainwindow.cc, mainwindow.ui* (joihin **EI SAA TEHDÄ *MITÄÄN* MUUTOKSIA)**

- Pääohjelma, joka hoitaa syötteiden lukemisen, komentojen tulkitsemisen ja tulostusten tulostamisen. Pääohjelmassa on myös valmiina komentoja testaamista varten.
- QtCreatorilla tai qmakella käännettäessä graafinen käyttöliittymä, jonka "komentotulkkiin" voi näppäimistön lisäksi hiirellä lisätä komentoja, tiedostoja yms. Graafinen käyttöliittymä näyttää myös luodut kaupungit ja niiden vasallisuhteet graafisesti samoin kuin suoritettujen operaatioiden tulokset.

Tiedosto *datastructures.hh*

- `class Datastructures`: Luokka, johon harjoitustyö kirjoitetaan. Luokasta annetaan valmiina sen julkinen rajapinta (johon **EI SAA TEHDÄ *MITÄÄN* MUUTOKSIA)**
- Tyyppimäärittely `TownID`, jota käytetään kaupungit yksilöivänä tunnisteena (samannimisiä kaupunkeja voi olla monta ja vaikka samoissa koordinaateissa, mutta jokaisella on eri id).
- Vakiot `NO_ID`, `NO_NAME` ja `NO_VALUE`, joita käytetään paluuarvoina, jos tietoja kysytään kaupungista, jota ei ole olemassa.

Tiedosto *datastructures.cc*

- Tänne luonnollisesti kirjoitetaan luokan operaatioiden toteutukset.
- Funktio `random_in_range`: Arpoo luvun annetulla välillä (alku- ja loppuarvo ovat molemmat välissä mukana). Voit käyttää tätä funktiota, jos tarvitset toteutuksessasi satunnaislukuja.

Graafisen käyttöliittymän käytöstä

Tässä harjoitustyössä graafisesta käyttöliittymästä (QtCreatorilla käännettäessä) on enemmän hyötyä testaamisessa, koska kaupunkien sijaintien lisäksi käyttöliittymä näyttää kaupunkien väliset verotussuhteet. Lisäksi käyttöliittymää on laajennettu niin, että kaupungin klikkaaminen hiirellä tulostaa sen tiedot (kuten ennenkin), mutta myös kopioi kaupungin ID:n komentoriville (kätevä tapa syöttää komentojen parametreja). Käyttöliittymästä saa valita, näyttääkö se graafisesti kaupungin nimet, vasallisuhteet ja tiet. Löytyneet reitit piirretään myös graafisesti (jos teiden piirto on päällä).

Harjoitustyönä toteutettavat osat

Tiedostot *datastructures.hh* ja *datastructures.cc*

- `class Datastructures`: Luokan julkisen rajapinnan jäsenfunktiot tulee toteuttaa. Luokkaan saa listätä omia määrittelyitä (jäsenmuuttujat, uudet jäsenfunktiot yms.)

- **Tiedostoon *datastructures.hh* kirjoitetaan jokaisen toteutetun operaation yläpuolelle kommentteihin oma arvio ko. operaation toteutuksen asymptoottisesti tehokkuudesta ja lyhyt perustelu arviolle.**

Lisäksi harjoitustyönä toteutetaan alussa mainittu dokumentti readme.pdf.

Huom! Omassa koodissa ei ole tarpeen tehdä ohjelman varsinaiseen toimintaan liittyviä tulostuksia, koska pääohjelma hoitaa ne. Mahdolliset Debug-tulostukset kannattaa tehdä cerr-virtaan (tai `qDebug`:lla, jos käytät Qt:ta), jotta ne eivät sotke testejä.

Ohjelman tuntemat komennot ja luokan julkinen rajapinta

Kun ohjelma käynnistetään, se jää odottamaan komentoja, jotka on selitetty alla. Komennot, joiden yhteydessä mainitaan jäsenfunktio, kutsuvat ko. Datastructure-luokan operaatioita, jotka siis opiskelijat toteuttavat. Osa komennoista on taas toteutettu kokonaan kurssin puolesta pääohjelmassa.

Jos ohjelmalle antaa komentoriviltä tiedoston parametriksi, se lukee komennot ko. tiedostosta ja lopettaa sen jälkeen.

Komento **`Julkinen jäsenfunktio`**	**Selitys**
add_town id nimi (x,y) vero `bool add_town(TownID id, std::string const& name, int x, int y, int tax);`	Lisää tietorakenteeseen uuden kaupungin annetulla uniikilla id:llä, nimellä, sijainnilla ja verotulolla. Aluksi kaupungit eivät ole minkään toisen kaupungin vasalleja. Jos annetulla id:llä on jo kaupunki, ei tehdä mitään ja palautetaan `false`, muuten palautetaan `true`.
remove id `bool remove_town(TownID id);`	Poistaa annetulla id:llä olevan kaupungin. Jos id ei vastaa mitään kaupunkia, ei tehdä mitään ja palautetaan `false`, muuten palautetaan `true`. Jos poistettavalla kaupungilla on vasallikaupunkeja ja isäntäkaupunki, poistettavan kaupungin vasallit siirtyvät poistettavan kaupungin isännän vasalleiksi. Jos poistettavalla ei ole isäntää, poistettavan mahdolliset vasallitkin jäävät ilman isäntää poiston jälkeen. *Tämän operaation tehokkuus ei ole kriittisen tärkeää (sitä ei oleteta kutsuttavan usein), joten se ei ole oletuksena mukana tehokkuustesteissä. Tämän operaation toteuttaminen ei ole pakollista (mutta otetaan huomioon arvostelussa).*
add_vassalship vassalid masterid `bool add_vassalship(TownID vassalid, TownID masterid);`	Lisää kaupunkien välille vasallisuhteen. Kaupunki voi olla vain yhden kaupungin vasallikaupunki. Työssä saa olettaa, että vasallisuhteet eivät voi muodostaa silmukoita (ts. kaupunki ei voi olla suoraan tai epäsuorasti itsensä vasalli). Jos jompaa kumpaa kaupunkia ei löydy tai vasallilla on jo isäntä, ei tehdä mitään ja palautetaan `false.` Muuten palautetaan `true`.

Komento `Julkinen jäsenfunktio`	Selitys
`(no command)` `std::string get_name(TownID id);`	Palauttaa annetulla ID:llä olevan kaupungin nimen, tai NO_NAME, jos id:llä ei löydy kaupunkia. (Pääohjelma kutsuu tätä eri paikoissa.) *Tätä operaatiota kutsutaan useammin kuin muita.*

`(no command)`
`std::pair<int, int>`
`get_coordinates(TownID id);`
	Palauttaa annetulla ID:llä olevan kaupungin sijainnin, tai parin (NO_VALUE, NO_VALUE), jos id:llä ei löydy kaupunkia. (Pääohjelma kutsuu tätä eri paikoissa.) *Tätä operaatiota kutsutaan useammin kuin muita.*

`(no command)` `int get_tax(TownID id);`	Palauttaa annetulla ID:llä olevan kaupungin verotiedon, tai NO_VALUE, jos id:llä ei löydy kaupunkia. (Pääohjelma kutsuu tätä eri paikoissa.) *Tätä operaatiota kutsutaan useammin kuin muita.*

vassals id
`std::vector<TownID>`
`get_vassals(TownID id);`
	Palauttaa annetulla ID:llä olevan kaupungin välittömien vasallien id:t (ts. mukaan *ei lasketa* vasallien vasalleja), tai vektorin jonka ainoa alkio on NO_ID, jos id:llä ei löydy kaupunkia. Paluuarvo on järjestettävä nousevan ID:n mukaiseen järjestykseen. (Pääohjelma kutsuu tätä eri paikoissa.)

size `unsigned int size();`	Palauttaa tietorakenteessa olevien kaupunkien lukumäärän.
clear `void clear();`	Tyhjentää tietorakenteet eli poistaa kaikki kaupungit (tämän jälkeen size palauttaa 0).

`find nimi`
`std::vector<TownID>`
`find_towns(std::string const& name);`
	Palauttaa kaupungit, joilla on annettu nimi tai tyhjän vektorin, jos sellaisia ei ole. Paluuarvo on järjestettävä nousevan ID:n mukaiseen järjestykseen. *Tätä operaatiota kutsutaan harvoin, eikä se ole oletuksena mukana tehokkuustestissä.*

`change_name id newname`
`bool change_town_name(TownID id,`
`std::string const& newname);`
	Muuttaa annetulla ID:llä olevan kaupungin nimen. Jos kaupunkia ei löydy, palauttaa `false`, muuten `true`.

all_towns `std::vector<TownID> all_towns();`	Palauttaa kaikki tietorakenteessa olevat kaupungit mielivaltaisessa järjestyksessä. *Tämä operaatio ei ole oletuksena mukana tehokkuustesteissä.*

alphalist
`std::vector<TownID>`
`towns_alphabetically();`
	Palauttaa kaupungit nimen mukaan aakkosjärjestyksessä.

distlist
`std::vector<TownID>`
`towns_distance_increasing();`
	Palauttaa kaupungit etäisyysjärjestyksessä origosta (0,0), lähin ensin. *Huomaa etäisyyden määritelmä aiempana.*

mindist `TownID min_distance();`	Palauttaa kaupungin, joka on lähinnä origoa (0,0). Jos tällaisia on useita, palauttaa jonkin niistä. Jos kaupunkeja ei ole, palautetaan NO_ID. *Huomaa etäisyyden määritelmä aiempana.*

Komento `Julkinen jäsenfunktio`	Selitys
maxdist `TownID max_distance();`	Palauttaa kaupungin, joka on kauimpana origosta (0,0). Jos tällaisia on useita, palauttaa jonkin niistä. Jos kaupunkeja ei ole, palautetaan NO_ID. *Huomaa etäisyyden määritelmä aiempana.*
nth_distance n `TownID nth_distance(unsigned int n);`	Palauttaa kaupungin, joka on etäisyysjärjestyksessä n:s origosta (0,0). Jos tällaisia on useita, palauttaa jonkin niistä. Jos *n* on 0 tai suurempi kuin kaupunkien määrä, palautetaan NO_ID. *Huomaa etäisyyden määritelmä aiempana.*
towns_from (x,y) `std::vector<TownID> towns_distance_increasing_from(int x, int y);`	Palauttaa kaupungit etäisyysjärjestyksessä annetusta koordinaatista (x,y), lähin ensin. *Huomaa etäisyyden määritelmä aiempana. Tämän operaation toteuttaminen ei ole pakollista (mutta otetaan huomioon arvostelussa).*
taxer_path id `std::vector<TownID> taxer_path(TownID id);`	Palauttaa listan kaupungeista, joille kaupunki maksaa veroa suoraan tai epäsuorasti. Paluuvektoriin talletetaan ensin kaupunki itse, sitten sen isäntä, isännän isäntä jne. niin kauan kuin isäntiä riittää. Jos id:llä ei ole kaupunkia, palautetaan tyhjä vektori.
longest_vassal_path id `std::vector<TownID> longest_vassal_path(TownID id);`	Palauttaa pisimmän mahdollisen ketjun vasalleja kaupungista lähtien. Paluuvektoriin talletetaan ensin kaupunki, sitten kaupungin vasalli, vasallin vasalli jne. niin, että ketjussa on mahdollisimman monta kaupunkia (jos yhtä pitkiä ketjuja on useita, palautetaan jokin niistä). Jos id:llä ei ole kaupunkia, palautetaan tyhjä vektori. *Tämän komennon toteutus ei ole pakollinen (mutta se vaikuttaa arvosteluun).*
total_net_tax id `int total_net_tax(TownID id);`	Palauttaa kaupungin nettoverokertymän (joka on määritelty tässä ohjeessa aiemmin). Jos id:llä ei löydy kaupunkia, palautetaan NO_VALUE. *Tämän komennon toteutus ei ole pakollinen (mutta se vaikuttaa arvosteluun).*
add_road id1 id2 `bool add_road(TownID town1, TownID town2)`	Lisää annettujen kaupunkien välille tien. Jos jompaa kumpaa kaupunkia ei löydy tai niiden välillä on jo tie, ei tehdä mitään ja palautetaan `false`, muuten `true`.
remove_road id1 id2 `bool remove_road(TownID town1, TownID town2)`	Poistaa annettujen kaupunkien väliltä tien. Jos jompaa kumpaa kaupunkia ei löydy tai niiden välillä ei ole tietä, ei tehdä mitään ja palautetaan `false`, muuten `true`.

Komento `Julkinen jäsenfunktio`	Selitys
all_roads **std::vector<std::pair<TownID, TownID>> all_roads()**	Palauttaa listan kaikista teistä. Jokainen tie on listassa pari kaupunki-id:itä, ja esitetään siinä järjestyksessä, että parin ensimmäinen id on pienempi kuin toinen id. Listassa parit on järjestetty ensisijaisesti ensimmäisen id:n mukaan, toissijaisesti toisen (std::pair:n oletusarvoinen <-vertailu). *Tämän operaation tehokkuutta ei mitata eikä arvostella, se on olemassa debuggausta ja testausta helpottamaan.*
roads id **std::vector<TownID> get_roads_from(TownID id)**	Palauttaa nousevan id:n mukaisessa järjestyksessä listan kapungeista, joihin annetusta kaupungista menee suoraan tie. Jos annettua kaupunkia ei löydy, palautetaan ainoana alkiona NO_ID.
clear_roads **void clear_roads();**	Poistaa kaikki tiet (mutta ei poista kaupunkeja). *Tämän operaation tehokkuutta ei mitata eikä arvostella, se on olemassa debuggausta ja testausta helpottamaan.*
any_route id1 id2 **std::vector<TownID> any_route(TownID fromid, TownID toid)**	Palauttaa jonkin (mielivaltaisen) reitin kaupungista id1 kaupunkiin id2. Palautetussa vektorissa on ensimmäisenä id1, sitten kaikki reitin varrella olevat kaupungit järjestyksessä ja viimeisenä id2. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jompaa kumpaa kaupunkia ei löydy, ainoana alkiona palautetaan NO_ID.
least_towns_route id1 id2 **std::vector<TownID> least_towns_route(TownID fromid, TownID toid)**	Palauttaa sellaisen reitin kaupungista id1 kaupunkiin id2, jonka varrella on mahdollisimman vähän kaupunkeja. Jos usealla reitillä on yhtä vähän kaupunkeja, palautetaan jokin niistä. Palautetussa vektorissa on ensimmäisenä id1, sitten kaikki reitin varrella olevat kaupungit ja viimeisenä id2. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jompaa kumpaa kaupunkia ei löydy, ainoana alkiona palautetaan NO_ID.
shortest_route id1 id2 **std::vector<TownID> shortest_route(TownID fromid, TownID toid)**	Palauttaa sellaisen reitin kaupungista id1 kaupunkiin id2, jonka kokonaispituus on mahdollisimman pieni (tien pituus on kaupunkien etäisyys käyttäen tässä työohjeessa olevaa etäisyyden määritelmää). Jos useat reitit ovat yhtä lyhyitä, palautetaan jokin niistä. Palautetussa vektorissa on ensimmäisenä id1, sitten kaikki reitin varrella olevat kaupungit ja viimeisenä id2. Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos jompaa kumpaa kaupunkia ei löydy, ainoana alkiona palautetaan NO_ID.

Komento `Julkinen jäsenfunktio`	Selitys
road_cycle_route id **std::vector<TownID>** **road_cycle_route(TownID startid)**	Tarkastaa, voiko annetusta kaupungista kulkea tieverkkoa pitkin niin, että reittiin muodostuu silmukka (palataan eri tietä johonkin reitin varrella olevaan kaupunkiin). Paluuarvona palautetaan reitti, joka päättyy ko. silmukkaan (paluuarvon viimeisenä kaupunkina on kaupunki, johon palaaminen muodostaa silmukan). Jos reittiä ei löydy, palautetaan tyhjä vektori. Jos annettua kaupunkia ei löydy, palautetaan ainoana alkiona NO_ID. *Huom, tulostuksen yhtenäistämiseksi pääohjelma tulostaa palautetusta reitistä vain silmukan, ja vielä niin, että silmukka kierretään haarautumiskohdasta pienemmän id:n suuntaan. Graafisesti reitti näytetään kokonaisuudessaan.*
trim_road_network **Dist trim_road_network()**	Jättää jäljelle tiet, joiden yhteenlaskettu pituus on mahdollisimman pieni, mutta joilla edelleen löytyy reitti sellaisten kaupunkien välillä, joiden välillä oli ennenkin reitti. Muut tiet poistetaan. Paluuarvona palautetaan tuloksena olevan tieverkoston kokonaispituus. Jos olemassa useita kokonaispituudeltaan yhtä lyhyitä tieverkostoja, mikä tahansa niistä kelpaa. *Tämä operaatio ei oletuksena ole mukana all-tehokkuustestissä, mutta sen voi valita tehokkuustestiin mukaan erikseen.*
random_add n (pääohjelman toteuttama)	Lisää tietorakenteeseen (testausta varten) *n* kpl kaupunkeja, joilla on satunnainen id, nimi, sijainti ja vero. 80 % todennäköisyydellä kaupunki lisätään myös toisen vasalliksi. Huom! Arvot ovat tosiaan satunnaisia, eli saattavat olla kerrasta toiseen eri.
random_roads n (pääohjelman toteuttama)	Lisää kaupunkien välille maksimissaan n satunnaista tietä niin, että arvotut tiet eivät risteä keskenään (käyttöliittymän selkeyttämiseksi). Teitä saatetaan lisätä myös vähemmän. *Tehokkuustestissä tämä komento lisää aina maks. 5 tietä.*
random_road_network (pääohjelman toteuttama)	Lisää kaupunkien välille satunnaisen tieverkoston, jossa jokaisesta kaupungista pääsee toiseen ja tiet eivät risteä keskenään.
random_seed n (pääohjelman toteuttama)	Asettaa pääohjelman satunnaislukugeneraattorille uuden siemenarvon. Oletuksena generaattori alustetaan joka kerta eri arvoon, eli satunnainen data on eri ajokerroilla erilaista. Siemenarvon asettamalla arvotun datan saa toistumaan samanlaisena kerrasta toiseen (voi olla hyödyllista debuggaamisessa).
read 'tiedostonimi' (pääohjelman toteuttama)	Lukee lisää komentoja annetusta tiedostosta. (Tällä voi esim. lukea listan tiedostossa olevia työntekijöitä tietorakenteeseen, ajaa valmiita testejä yms.)

Komento **`Julkinen jäsenfunktio`**	**Selitys**
stopwatch on / off / next (pääohjelman toteuttama)	Aloittaa tai lopettaa komentojen ajanmittauksen. Ohjelman alussa mittaus on pois päältä ("off"). Kun mittaus on päällä ("on"), tulostetaan jokaisen komennon jälkeen siihen kulunut aika. Vaihtoehto "next" kytkee mittauksen päälle vain seuraavan komennon ajaksi (kätevää read-komennon kanssa, kun halutaan mitata vain komentotiedoston kokonaisaika).
perftest all/compulsory/cmd1;cmd2... **timeout n n1;n2;n3...** (pääohjelman toteuttama)	Ajaa ohjelmalle tehokkuustestit. Tyhjentää tietorakenteen ja lisää sinne *n1* kpl satunnaisia kaupunkeja (ks. random_add) ja näiden välille satunnaisen määrän teitä. Sen jälkeen arpoo *n* kertaa satunnaisen komennon. Mittaa ja tulostaa sekä lisäämiseen että komentoihin menneen ajan. Sen jälkeen sama toistetaan *n2*:lle jne. Jos jonkin testikierroksen suoritusaika ylittää *timeout* sekuntia, keskeytetään testien ajaminen (tämä ei välttämättä ole mikään ongelma, vaan mielivaltainen aikaraja). Jos ensimmäinen parametri on *all*, arvotaan lisäyksen jälkeen kaikkista komennoista, joita on ilmoitettu kutsuttavan usein. Jos se on *compulsory*, testataan vain komentoja, jotka on pakko toteuttaa. Jos parametri on lista komentoja, arvotaan komento näiden joukosta (tällöin kannattaa mukaan ottaa myös random_add, jotta lisäyksiä tulee myös testikierroksen aikana). Jos ohjelmaa ajaa graafisella käyttöliittymällä, "stop test" nappia painamalla testi keskeytetään (nappiin reagointi voi kestää hetken). *Huom! 3. harjoitustyössä perftest ei oletuksena enää testaa 1. ja 2. työn aakkostamiseen ja verotukseen liittyviä operaatioita, ainoastaan kaupunkien lisäämisiä, erityisen usein tehtäviä operaatioita ja teihin liittyviä uusia operaatioita (lisäämisvaiheessa lisätään edelleen satunnaisia vasallisuhteita).*
testread 'in-tiedostonimi' 'out-tiedostonimi' (pääohjelman toteuttama)	Ajaa toiminnallisuustestin ja vertailee tulostuksia. Lukee komennot tiedostosta in-tiedostonimi ja näyttää ohjelman tulostuksen rinnakkain tiedoston out-tiedostonimi sisällön kanssa. Jokainen eroava rivi merkitään kysymysmerkillä, ja lopuksi tulostetaan vielä tieto, oliko eroavia rivejä.
help (pääohjelman toteuttama)	Tulostaa listan tunnetuista komennoista.
quit (pääohjelman toteuttama)	Lopettaa ohjelman. (Tiedostosta luettaessa lopettaa vain ko. tiedoston lukemisen.)

Tehokkuustestit

Tässä harjoitustyössä suuri osa operaatioista on ei-pakollisia, joten yhtä kattavaa tehokkuustestiä on mahdotonta antaa. Sen vuoksi koodihakemistosta löytyy useita perftest-alkuisia testejä. Niistä yksi testaa vain pakollisia osia, loput testaavat kukin vain yhtä ei-pakollista operaatiota. Huomatkaa, että tässä harjoitustyössä asymptoottiseen tehokkuuteen ei välttämättä voi kovin paljoa vaikuttaa, koska jotkin algoritmit vain ovat luonteeltaan raskaita.

"Datatiedostot"

Kätevin tapa testata ohjelmaa on luoda "datatiedostoja", jotka add-komennolla lisäävät joukon kaupunkeja ohjelmaan. Kaupungit voi sitten kätevästi lukea sisään tiedostosta read-komennolla ja sitten kokeilla muita komentoja ilman, että kaupungit täytyisi joka kerta syöttää sisään käsin. Alla on esimerkki datatiedostosta, joka löytyy nimellä *example-data.txt*:

```
# Adding towns
add_town Hki Helsinki (3,0) 3
add_town Tpe Tampere (2,2) 4
add_town Ol Oulu (3,7) 10
add_town Kuo Kuopio (6,3) 9
add_town Tku Turku (1,1) 2
# Adding crossroads as extra towns
add_town x1 xx (3,3) 6
add_town x2 xy (4,4) 8
# Adding roads
add_road Tpe x1
add_road x1 x2
add_road x2 Ol
add_road Ol Kuo
add_road Tpe Kuo
add_road Hki Tpe
add_road Tpe Tku
```

Esimerkki ohjelman toiminnasta

Alla on esimerkki ohjelman toiminnasta. Esimerkin syöte löytyy tiedostoista *example-...-in.txt* ja tulostukset tiedostoista *example-...-out.txt.* Testaamista varten on myös tiedosto *all-examples_in.txt*, joka lukee kaikki esimerkkitiedostot testread-komennolla ja vertaa niitä tulostuksiin (samanlainen tiedosto löytyy myös muille testityypeille).

- *example-compulsory-out.txt*

```
> clear
Cleared all towns
> read "example-data.txt"
** Commands from 'example-data.txt'
...
** End of commands from 'example-data.txt'
> all_roads
1: Hki <-> Tpe (3)
2: Kuo <-> Ol (7)
3: Kuo <-> Tpe (5)
```

```
4: Ol <-> x2 (4)
5: Tku <-> Tpe (2)
6: Tpe <-> x1 (2)
7: x1 <-> x2 (2)
> roads Tpe
1. Helsinki: pos=(3,0), tax=3, id=Hki
2. Kuopio: pos=(6,3), tax=9, id=Kuo
3. Turku: pos=(1,1), tax=2, id=Tku
4. xx: pos=(3,3), tax=6, id=x1
> any_path Tku Hki
0.    0: Turku: pos=(1,1), tax=2, id=Tku
1. -> 2: Tampere: pos=(2,2), tax=4, id=Tpe
2. -> 5: Helsinki: pos=(3,0), tax=3, id=Hki
> remove_road Tpe Hki
Removed road: Tampere <-> Helsinki
> roads Tpe
1. Kuopio: pos=(6,3), tax=9, id=Kuo
2. Turku: pos=(1,1), tax=2, id=Tku
3. xx: pos=(3,3), tax=6, id=x1
> any_path Tku Hki
> clear_roads
All roads removed.
> all_roads
No roads!
```

- *example-least_towns_route-out.txt*

```
> clear
Cleared all towns
> read "example-data.txt"
** Commands from 'example-data.txt'
...
** End of commands from 'example-data.txt'
> all_roads
1: Hki <-> Tpe (3)
2: Kuo <-> Ol (7)
3: Kuo <-> Tpe (5)
4: Ol <-> x2 (4)
5: Tku <-> Tpe (2)
6: Tpe <-> x1 (2)
7: x1 <-> x2 (2)
> least_towns_route Hki Ol
0.    0: Helsinki: pos=(3,0), tax=3, id=Hki
1. -> 3: Tampere: pos=(2,2), tax=4, id=Tpe
2. -> 8: Kuopio: pos=(6,3), tax=9, id=Kuo
3. -> 15: Oulu: pos=(3,7), tax=10, id=Ol
```

- *example-shortest_route-out.txt*

```
> clear
Cleared all towns
> read "example-data.txt"
** Commands from 'example-data.txt'
...
** End of commands from 'example-data.txt'
```

```
> all_roads
1: Hki <-> Tpe (3)
2: Kuo <-> Ol (7)
3: Kuo <-> Tpe (5)
4: Ol <-> x2 (4)
5: Tku <-> Tpe (2)
6: Tpe <-> x1 (2)
7: x1 <-> x2 (2)
> shortest_route Hki Ol
0.    0: Helsinki: pos=(3,0), tax=3, id=Hki
1. -> 3: Tampere: pos=(2,2), tax=4, id=Tpe
2. -> 5: xx: pos=(3,3), tax=6, id=x1
3. -> 7: xy: pos=(4,4), tax=8, id=x2
4. -> 11: Oulu: pos=(3,7), tax=10, id=Ol
```

- *example-road_cycle_route-out.txt*

```
> clear
Cleared all towns
> read "example-data.txt"
** Commands from 'example-data.txt'
...
** End of commands from 'example-data.txt'
> all_roads
1: Hki <-> Tpe (3)
2: Kuo <-> Ol (7)
3: Kuo <-> Tpe (5)
4: Ol <-> x2 (4)
5: Tku <-> Tpe (2)
6: Tpe <-> x1 (2)
7: x1 <-> x2 (2)
> road_cycle_route Tku
0.    Tampere: pos=(2,2), tax=4, id=Tpe
1. -> Kuopio: pos=(6,3), tax=9, id=Kuo
2. -> Oulu: pos=(3,7), tax=10, id=Ol
3. -> xy: pos=(4,4), tax=8, id=x2
4. -> xx: pos=(3,3), tax=6, id=x1
5. -> Tampere: pos=(2,2), tax=4, id=Tpe
> remove_road Ol Kuo
Removed road: Oulu <-> Kuopio
> road_cycle_route Tku
No road cycles found.
```

- *example-trim_road_network-out.txt*

```
> clear
Cleared all towns
> read "example-data.txt"
** Commands from 'example-data.txt'
...
** End of commands from 'example-data.txt'
> all_roads
1: Hki <-> Tpe (3)
2: Kuo <-> Ol (7)
3: Kuo <-> Tpe (5)
```

```
4: Ol <-> x2 (4)
5: Tku <-> Tpe (2)
6: Tpe <-> x1 (2)
7: x1 <-> x2 (2)
> trim_road_network
The remaining road network has total distance of 18
> all_roads
1: Hki <-> Tpe (3)
2: Kuo <-> Tpe (5)
3: Ol <-> x2 (4)
4: Tku <-> Tpe (2)
5: Tpe <-> x1 (2)
6: x1 <-> x2 (2)
```